Design and Performance Analysis of CAPE based on Discontinuous Incremental Checkpoints

Viet Hai Ha, Eric Renault

Design and Performance Analysis of CAPE based on Discontinuous Incremental Checkpoints

Viet Hai Ha and Éric Renault
Institut Télécom – Télécom SudParis
Samovar UMR INT-CNRS 5157
Évry, France
viet_hai.ha@it-sudparis.eu and eric.renault@it-sudparis.eu

***Abstract*—Checkpointing Aided Parallel Execution (CAPE) is a paradigm using checkpointing technique to distribute sequential programs equipped with OpenMP directives in distributed systems. In its first prototype, the use of a complete checkpointer strongly decreased global performance. This paper shows how the performance of the CAPE paradigm have been improved using discontinuous incremental checkpointing and provide an in-depth analysis of this performance.**

I. Introduction

The architecture of parallel machines has drastically changed over the past fifteen years, from mainframes that were the typical solution for parallel computing at the beginning of the 90's to the clusters [1] at the end of the 90's or more recently to grids [2]. Available tools to develop applications on top of these platforms also have evolved but differently. As two main architectures have been identified for parallel machines, shared-memory architectures on the one hand and distributed-memory architecture on the other hand, specific tools have been developed to cope with their specificities. One is OpenMP [3], developed for shared-memory machines and PVM, and the other is MPI [4], for distributed-memory systems. If the programming paradigm associated with OpenMP is quite simple to handle by programmers for its similarities with the more traditional sequential paradigm, the message-passing paradigm associated with distributed-memory system is more difficult to convince a large public to use it. As a result, there have been several attempts to develop a compiler that automatically generates a version of OpenMP programs that is capable of running on a distributed-memory architecture. They can be divided into two categories.

The first category consists in using a Single System Image (SSI) in order to hide the distributed nature of the underlying architecture to the parallel application [5]. An SSI aims at providing an abstraction layer on top of a distributed system so that users and applications can use resources as they were parts of a single monolithic machine. Typically, this means that the set of processors is seen as an SMP or a single multicore processor; the set of available RAMs in each node is seen as a single memory; and the set of file systems is accessible as a single one. Several SSI are now available and mature with different capabilities and targeting different architectures. Some solutions for clusters are Genesis [6], Millipede [7], Nomad [8] and Kerrighed [9]. A newer solution for grids is XtreemOS [10], a derivative product from Kerrighed. XtreemOS is still an ongoing work and also targeting clouds [11]. The main advantage of using an SSI to run an OpenMP program is that the program can run as is, with no need to recompile if an executable file is already available.

The second category consists in using a parallel library and translating OpenMP directives and memory updates to call to the parallel library functions. Two solutions have emerged as of today, one developed on top of MPI [12] and the other one on top of Global Arrays [13]. The main criticism towards solutions based on a parallel library is that they have lots of difficulties to take into account all memory accesses. Typically, when a memory location is accessed through a differentiation or a set of differentiation, it is sometimes very difficult for the compiler to identify the data type associated to the memory area.

Apart from the above two categories, we have been developing CAPE (which stands for Checkpointing Aided Parallel Execution) [14] [15], a method that uses checkpointing technique to execute OpenMP programs on distributed architectures. The first prototype of CAPE proved the feasibility of the approach but the use of a complete checkpointer as the base tool strongly decreased the global performance. This article aims at showing how the performance of the CAPE paradigm have been improved using the discontinuous incremental checkpointer we developed and at providing an in-depth analysis of this performance.

The article is organized as follows: section II presents the principles inherent to CAPE. The next one focusses on the discontinuous checkpointer and develops how these principles have been modified to cope with incremental checkpoints. The last section before the conclusion is dedicated to the in-depth analysis of the performance measurements.

II. CAPE principles

CAPE, which stands for Checkpointing Aided Parallel Execution, aims at automatically transforming a parallel shared-memory program so that it can be executed on a distributed-memory architecture. In the current implementation that we developed, CAPE is able to handle OpenMP directives provided in a C programming language program. However, CAPE is not language dependent and could be extended to any programming language and/or parallel library or tool.

The basic idea of CAPE is that while many researches have been conducted in order to develop checkpointing applications to save the state of a program, CAPE makes use of checkpoints in order to allow programs to run on a distributed-memory architecture instead of a shared-memory architecture. The main difference between these two architectures remains in the fact that two segments out of three (both text and data segments vs. the stack) are shared by the different threads belonging to a single parallel process while the two processes belonging to a single parallel application are executing in two completely different memory address spaces. For both architectures, stacks belonging to different threads are stored at different locations. And as a thread should not access the private data of another directly, there should be no portability problem from a shared-memory architecture to a distributed-memory architecture. As most programs executing on distributed-memory architectures are SPMD, the text segment which is read-only by definition is not changed during execution and is therefore consistent among all the nodes. Thus, this segment involves no problem either. The situation is different for the data segment which memory locations may be accessed by any thread at any time. In the scope of CAPE, the virtual address space is taken into account as a whole and no difference is made among the different segments.

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ automatically translated into ↓

```
parent = create ( original )
if ( ! parent )
    exit
copy ( original, target )
for ( A ; B ; C )
    parent = create ( before_i )
    if ( parent )
        ssh host_x restart ( before_i )
    else
        D
        parent = create ( after_i )
        if ( ! parent )
            exit
        diff ( before_i, after_i, delta_i )
        merge ( target, delta_i )
        exit
parent = create ( final )
if ( parent )
    diff ( original, final, delta )
    wait_for ( target )
    merge ( target, delta )
    restart ( target )
```

Fig. 1. Template for OpenMP with complete checkpoints.

Figure 1 presents the effective transformation that is performed on a code that specifies a parallel `for` that has all loop iterations D satisfy Bernstein's conditions using OpenMP directives. The parent, i.e. the master node, is in charge of managing the slaves only and does not execute any loop iteration in the parallel part. However, this is not mandatory and the master node could also take part in the execution of one or more loop iterations. The translation is based on the following functions:

- `create` creates a checkpoint and saves it in the file provided as a parameter. The value returned by the function is used to identify whether the function has just created the checkpoint and returned, or the process has been created after resuming the execution from the checkpoint. This function is very similar to the `fork` system call, except that `create` returns `TRUE` after generating the checkpoint and `FALSE` after resuming the execution from the checkpoint.
- `copy` copies a file into another one.
- `diff` saves into the last file provided as a parameter the list of modifications that should be applied on the first file to obtain the second one.
- `merge` applies the list of modifications saved in the second file provided as a parameter to the checkpoint file provided as the first parameter.
- `wait_for` returns after the file whose the name is provided as a parameter is available.
- `restart` resumes the execution of the current process from the checkpoint file provided as a parameter.

Note that the operation that consists in resuming the execution of the checkpoints generated for each loop iteration, the line in italic in Fig. 1, is executed on the master node but delegated to an external process in charge of managing the distribution of processes on a set of remote resources. BOINC [16], used in the scope of the Seti@Home project, is probably one of the most famous tool aiming at distributing works among a set of computing resources. For an in-depth description of CAPE, refer to [14] and [15].

III. DICKPT AND A NEW MODEL FOR CAPE

The performance analysis of the implementation of CAPE based on a complete checkpointer showed that an important part of the program execution is spent in creating checkpoints, sending checkpoints over the network, computing the difference between two checkpoints, and injecting the previously computed difference into a process [17]. An optimization had been introduced by distributing the computation of the differences between two checkpoints on the set of nodes and then return to the master node the difference only instead of the complete checkpoint, but performance results still remained quite poor as at least one complete checkpoint had to be sent over the network. It clearly appeared that the unique viable solution consists in using incremental checkpoints only.

The main idea behind using incremental checkpoints is twofold: first, this allows to transmit far less data over the

network; second, the time needed to deal with incremental checkpoints is more interesting. For example, instead of creating a checkpoint and then compute the differences between this new checkpoint and another one that serves as a reference, it is now possible to directly generate the set of differences as it is the checkpoint itself. Moreover, the use of incremental checkpoints also allows to avoid the copy of complete checkpoints that is time consuming.

Despite the availability of several incremental checkpointers, we decided to implement our own one in order to make sure it perfectly matches our needs [18]. In fact, the implementation of CAPE based on incremental checkpoints requires the ability to suspend and resume the checkpointer. Thus, we developed DICKPT [18] (which stands for DIscontinuous ChecKPoinTing) that allows to start and stop checkpointing at any location in the program. It is based on a buffer and a set of three primitives. The buffer aims at storing all the modifications that occurred on the process since the last time the checkpointer started or resumed its execution. These three primitives behave as follows:

- `start` clears the buffer and then starts or resumes checkpointing. Any modifications occurring on the process after the call to `start` are reported in the buffer. A call to `start` while the checkpointer is active results in clearing the content of the buffer which is definitively lost.
- `stop` stops checkpointing, i.e. any modifications that occurs on the process after the call to `stop` is not reported in the buffer. A call to `stop` while the checkpointer is not active is just discarded.
- `create` saves the content of the buffer in the file provided as a parameter. Several calls to `create` may occur inside a `start`/`stop` pair. In this case, the buffer containing the modifications that have been performed on the process is reinitialized for each call. There are two sub cases depending on the name of created files. The first case occurs when the file name matches the one of previous call. In this case the new file will be merged with the existed file. In the other case, the new file will be independently created. Function `create` may be called while the checkpointer is active.

For the rest of the paper, the meaning of function `create` is the one above.

Figure 2 presents the new version of the piece of code that is substituted to an OpenMP parallel `for` construct. The semantic associated to functions `merge` on this figure are exactly the same as the one presented for Fig. 1. The other functions are defined as follows:

- `master` returns `TRUE` when executing on the master node and `FALSE` when executing on a slave.
- `last_parallel` returns `TRUE` when the current parallel block is the last one of the entire program and `FALSE` otherwise.
- `send` transfers the content of the file provided as the first parameter to the node provided as the second parameter.

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ automatically translated into ↓

```
1   if ( master ( ) )
2       start ( )
3       for ( A ; B ; C )
4           create ( before )
5           send ( before, slave_x )
6       create ( final )
7       stop ( )
8       wait_for ( after )
9       inject ( after )
10      if ( ! last_parallel ( ) )
11          merge ( final, after )
12          broadcast ( final )
13  else
14      receive ( before )
15      inject ( before )
16      start ( )
17      D
18      create ( after_i )
19      stop ( )
20      send ( after_i, master )
21      if ( ! last_parallel ( ) )
22          receive ( final )
23          inject ( final )
24      else
25          exit
```

Fig. 2. Template for OpenMP with incremental checkpoints.

- `broadcast` sends a file to all the slaves. This function can only be executed on the master node.
- `receive` waits for the file provided as a parameter to be available.
- `inject` updates the current process with the information provided in the checkpoint file provided as a parameter. Note that this function does not update the instruction pointer.
- `wait_for` waits and merges all the components of the file provided as a parameter.

Two assumptions have been made to make the template works. The first one is that the platform for the master node and the slaves are homogeneous. This is easy to achieve, especially today with the rapid growth of virtualization. The second assumption states that no slave process has interactions with its environment. This second assumption can easily be handled by intercepting the calls to system calls and returning the result of the execution of the system call on the master node. In fact, if the master node is the only one to execute system calls, it becomes easy to detect whether a system call had already been executed and thus avoid its execution the

second time.

Apart from the use of incremental checkpoints instead of complete checkpoints, one of the most noticeable improvements between the original template and the one in Fig. 2 is that the process is never restarted. Checkpoints, which are far lighter than in the previous case, can only be used to `inject` the differences into a process and cannot be used to restart the process directly. Also note that this new template can be applied several times one after another inside a single program, or can be nested.

IV. Performance evaluation

In order to validate our approach, some performance measurements have been conducted on a Desktop Grid. This testbed is composed of nodes including Intel(R) Core(TM)2 Duo E8400 CPUs running at 3 GHz and 2 GB RAM, operated by Linux kernel 2.6.35 with Ubuntu 10.10 flavour, and connected by a standard Ethernet. In order to avoid as much as possible external influences, the entire system was dedicated to the tests during performance measurements.

The program used for tests is a matrix-matrix product for which the size varies from 3,000×3,000 to 12,000×12,000. Matrices are supposed to be dense and no specific algorithm has been implemented to take into account sparse matrices. Each experiment has been performed at least 10 times and a confidence interval of at least 90% has always been achieved for the measures. Data reported here are the means of the 10 measures.

Size	Sequential	OpenMP
3,000	258.9	142.4
6,000	1,852.7	1,048.7
9,000	7,314.5	3,986.2
12,000	14,990.5	8,999.4

TABLE I
Execution time (in seconds) on a single node.

The execution of both the sequential version and the OpenMP version of the program on one of the nodes gives the result provided in Table I. A single core was used for the sequential execution of the program, while the OpenMP program took benefits of the two cores. One can check that results in the Table I are consistent as the execution time for both sequential and OpenMP versions are directly proportional to the cube of the matrix size. Typically, this means that no important cache effects have polluted the performance measurements, probably because almost all data were fitting into memory. Moreover, the speed-up obtained by OpenMP is 1.8 for the first three matrix sizes and 1.65 for the fourth one, which are expected values.

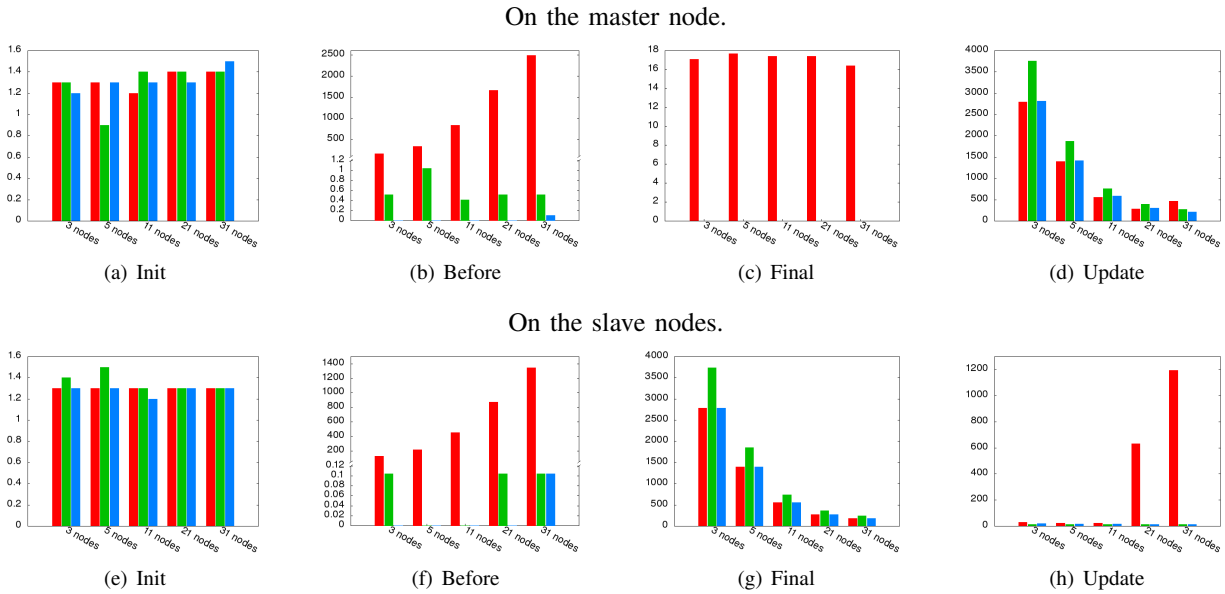
Figure 3 and 4 present the execution time in seconds of the matrix-matrix program for various number of nodes and matrix sizes. Note that, despite the fact that processors are dual core, a single core was used during the experiments. Three measures are represented each time: the left one is associated with CAPE using complete checkpoints, the middle one is also associated with CAPE but with incremental checkpoints, and the right one is associated with MPI. The MPI program has been developed for reference as exchanges to keep all processes consistent between nodes are kept minimal.

For both figures, two series of graphs are provided. The upper series is related to the master node, while the lower series is associated with the slave nodes. Each series is composed of four graphs:

- *Init* is the elapsed time between the beginning of the program and the beginning of the parallel for loop in the matrix-matrix product. On Fig. 2, these are all lines before the first one.
- *Before* is the time spent to create and send the checkpoints (lines 2 to 5) on the master. On slave nodes, this includes receiving and updating the slave process using the checkpoint (lines 14 and 15). For the specific case of MPI, this is the time to send data to slave nodes.
- *Final* is the time to generate the last checkpoint on the master node (lines 6 and line 7) and the time to do the job on the slaves (line 16 and line 17).
- *Update* is the time to receive all updates from the slave nodes and inject them in the master node (lines 8 and line 9). On slave nodes, this is the time to generate the incremental checkpoints and send them to the master node (lines 18 to 20). For the specific case of MPI, this is the time to send data to the master node.

Figure 3 presents the execution time for different number of nodes. The size of matrices are 9,000×9,000. However, similar trends are observed for the other matrix sizes. One can remark that the 3-node case apart, the execution time when using incremental checkpoints is always better than the execution time when using complete checkpoints. The larger the number of nodes, the smaller the execution time for both CAPE using incremental checkpoints and MPI. Moreover, the execution time for CAPE using incremental checkpoints is getting closer and closer as the number of nodes is increasing. The case for CAPE using complete checkpoints is different. When few nodes are used for the computation (up to 11), the execution time is decreasing as the number of nodes is increasing and the value is quite similar to the other two cases (CAPE using incremental checkpoints and MPI). However, for larger number of nodes, the execution time for CAPE using complete checkpoints is directly proportional to the number of nodes. This is due to the time needed to generate the checkpoints to be sent and the time to send these checkpoints over the network (there is at least one complete checkpoint for each slave node). This clearly justifies the use of incremental checkpoints for CAPE.

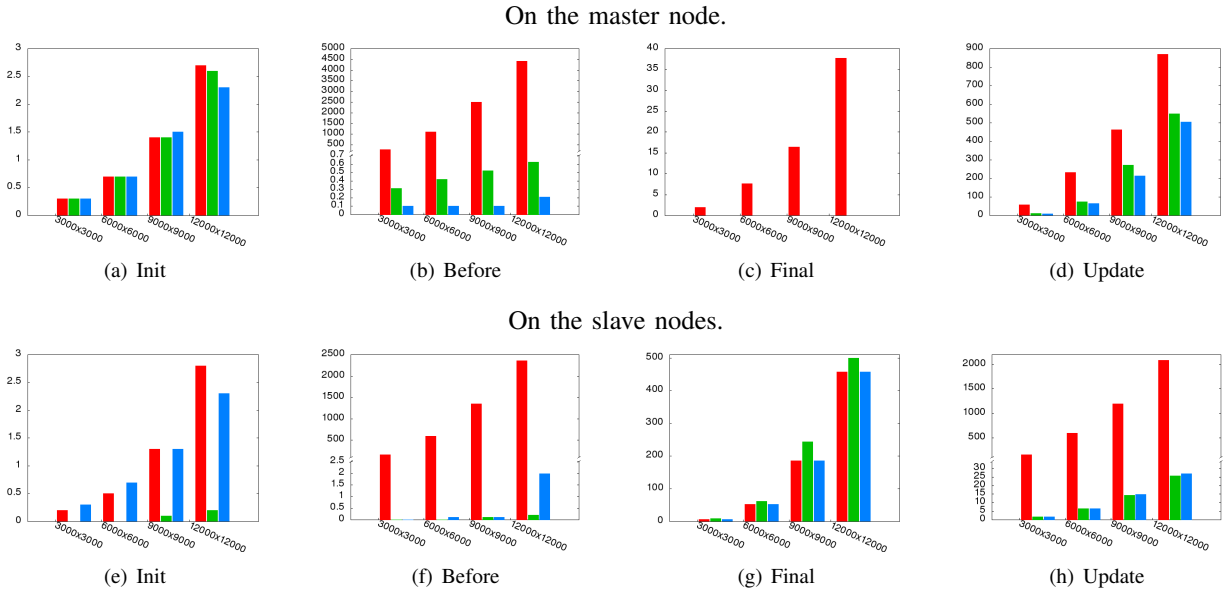
At first, the performance for three nodes may look strange as the execution time of the program with CAPE using complete checkpoints is better than the execution time with CAPE using incremental checkpoints. In fact, for small number of nodes, the amount of data transmitted over the network between the different nodes is almost the same for both complete and incre-



(a) Init (b) Before (c) Final (d) Update

(e) Init (f) Before (g) Final (h) Update

Fig. 3. Execution time (in seconds) vs. number of nodes.



(a) Init (b) Before (c) Final (d) Update

(e) Init (f) Before (g) Final (h) Update

Fig. 4. Execution time (in seconds) vs. problem size.

mental checkpoints as in the case of incremental checkpoints slave nodes receive a big part of matrices. However, in the case of incremental checkpoints, processes are monitored in order to capture the memory pages that are accessed for writing. The monitoring of the slave processes involves a computing overhead that is reduced proportionally with the amount of computation, and therefore with the number of nodes, when a large number of nodes is used. Fortunately, this is not a problem for CAPE. Processors with 4 and even 8 cores are available on the market and, as a result, CAPE is targeting architectures with a larger number of nodes.

Figure 4 presents the execution time for difference matrix sizes. The number of nodes involved in the parallel machine is 31. However, the remarks below would be the same with other number of nodes. The figure clearly shows that the execution time for CAPE using complete checkpoints is directly proportional to the square of the matrix size, while the execution time for both CAPE using incremental checkpoints and MPI is directly proportional to the matrix size. This is due to the fact that the virtual address space of the processes is mainly composed of the matrices, and that the complete virtual address space is transmitted over the network for complete checkpoints. However, for CAPE using incremental checkpoints and MPI, the complete virtual address spaces are

not transmitted over the network and only the data that have been updated during the computation of the matrix-matrix product are considered. Moreover, one can remark that the execution time for CAPE using incremental checkpoints and MPI are usually very close. This in-depth analysis of the performance results shows that globally the execution time for CAPE using incremental checkpoints is only 10% higher than the execution time for MPI, excepts for 3,000×3,000 matrices where the ratio is 1.3.

Note that graphs (c) on Fig. 3 and 4 do not show any data for CAPE with incremental checkpoints and MPI as the execution time for both is too small to be represented.

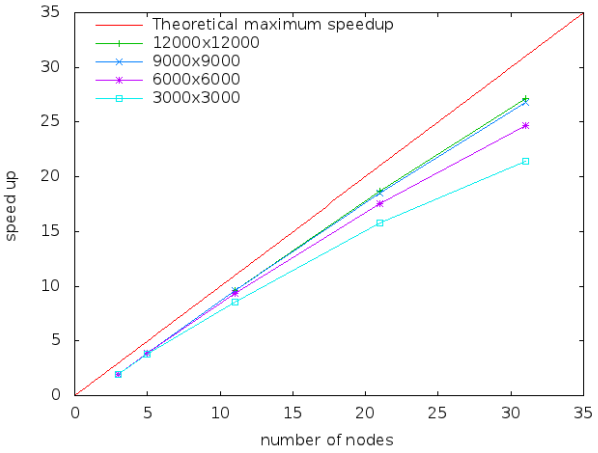


Fig. 5. Speedup vs. number of nodes.

Figure 5 shows the speedup of CAPE using incremental checkpoints for various number of nodes and matrix sizes. The red line represents the theoretical maximum speedup. The figure clearly shows that the solution is efficient with an efficiency (the ratio of the speedup over the number of nodes) in the range from 75% to 90%. Also, it highlights that the larger the size of the matrices, the higher the speedup, which was not the case with the complete checkpoint implementation.

V. Conclusion and Future Works

This article presented CAPE and more specifically the modifications that have been applied on the template algorithm to translate automatically parallel programs with OpenMP directives into a parallel program targeted for distributed-memory architectures together with the discontinuous incremental checkpointer we developed. An in-depth performance analysis is also provided that shows the legitimation of the incremental checkpointing approach.

At present, CAPE has proven its efficiency for the generation of code satisfying the Bernstein's conditions for distributed-memory architecture. In the near future, we have planed to go further the Bernstein's conditions and take into account shared variables.

References

[1] http://www.beowulf.org/

[2] Ian Foster, Carl Kesselman and Steven Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*. International Journal of Supercomputer Applications, 15(3):200–222, 2001.

[3] http://openmp.org/

[4] http://www.mpi-forum.org/

[5] Sven Karlsson, Sung-Woo Lee, Mats Brorsson, Sahni Sartaj, Viktor K. Prasanna and Shukla Uday. *A fully compliant OpenMP implementation on software distributed shared memory*. Proceedings of the International Conference on High Performance Computing, Bangalore, India, LNCS 2552, pp. 195–206, December 2002.

[6] Andrzej M. Goscinski, Michael Hobbs and Jack Silcock. *GENESIS: an efficient, transparent and easy to use cluster operating system*. Journal of Parallel Computing, 28(4):557–606, April 2002.

[7] Roy Friedman, Maxim Goldin, Ayal Itzkovitz and Assaf Schuster. *MILLIPEDE: Easy parallel programming in available distributed environments*. Journal of Software Practice and Experience, 27(8):929–965, 1997.

[8] Eduardo Pinheiro and Ricardo Bianchini. *Nomad: A scalable operating system for clusters of uni and multiprocessors*. Proceedings of the 1st IEEE International Workshop on Cluster Computing, Melbourne, Australia, pp. 247–254, December 1999.

[9] Christine Morin, Renaud Lottiaux, Geoffroy Valle, Pascal Gallard, Gal Utard, R. Badrinath and Louis Rilling. *Kerrighed: A Single System Image Clustet Operating System for High Performance Computing*. Euro-Par 2003 Parallel Processing, Klagenfurt, Austria, LNCS 2790, pp. 1291–1294, August 2003.

[10] Christine Morin. *XtreemOS: a Grid Operating System Making your Computer Ready for Participating in Virtual Organizations*. Proceedings of the IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), Santorini Island, Greece, Invited talk, May 2007.

[11] Contrail: Open Computing Infrastructure for Elastic Services. Web Site. `http://contrail-project.eu/`

[12] Ayon Basumallik and Rudolf Eigenmann. *Towards automatic translation of OpenMP to MPI*. Proceedings of the 19th annual international conference on Supercomputing, Cambridge, MA, pp. 189–198, 2005.

[13] Lei Huang and Barbara Chapman and Zhenying Liu. *Towards a more efficient implementation of OpenMP for clusters via translation to global arrays*. Journal of Parallel Computing, 31(10–12):1114–1139, October–December 2005.

[14] Éric Renault. *Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution*. International Workshop on OpenMP (IWOMP), Beijing, China, LNCS 4935, pp. 183–193, June 2007.

[15] Éric Renault. *Parallelization of For Loops Using Checkpointing Techniques*. Proceedings of the 2005 International Conference on Parallel Processing Workshops, Oslo, Norway, pp. 313–319, June 2005.

[16] David P. Anderson. *BOINC: A System for Public-Resource Computing and Storage*. Proceedings of 5th IEEE/ACM International Workshop on Grid Computing, Pittsburg, PA, pp. 4–10, November 2004.

[17] Laura Mereuta and Éric Renault. *Checkpointing Aided Parallel Execution Model and Analysis*. High Performance Computation Conference (HPCC), Houston, TX, LNCS 4782, pp. 707–717, September 2007.

[18] Viet Hai Ha and Éric Renault. *Discontinuous Incremental: A New Approach Towards Extremely Lightweight Checkpoints*. Proceedings of the International Symposium on Computer Networks and Distributed Systems (CNDS), Tehran, Iran, February 2011.